Lecture 09: Semaphores

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture

```c
int main() {
    int fd[2], status;
    pipe(fd);
    if(fork() == 0) {
        /* Child process */
        close(fd[0]);
        char buff[] = "Hello my dear good Parent";

        write(fd[1], buff, sizeof(buff));
        exit(0);
    }
    /* Parent process */
    close(fd[1]);
    char buff[100];
    read(fd[0], buff, sizeof(buff));
    printf("My obedient child says: %s\n", buff);
    wait(NULL);
    return 0;
}
```

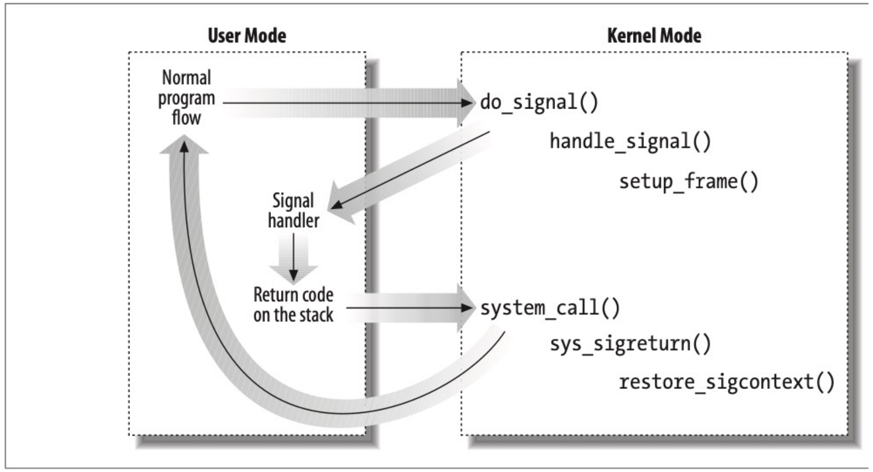
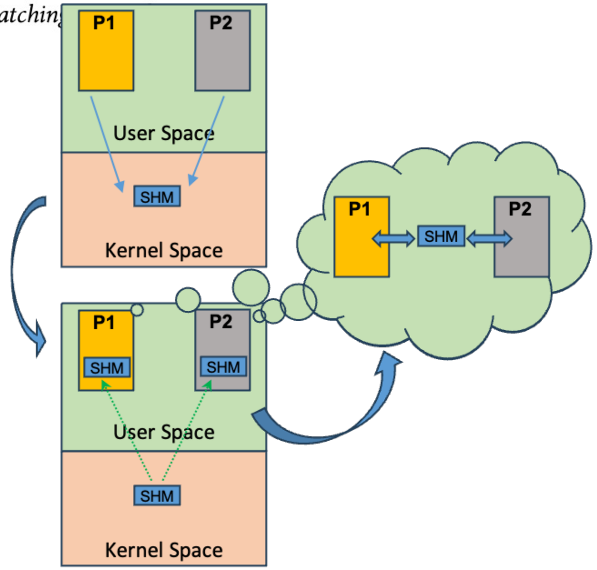


Figure 11-2. Catchin



Today's Class

- Semaphores for process synchronization
  - Mutual exclusion
  - Critical section
- Producer consumer problem

Array Sum Program (Version 1)

```
typedef struct shm_t {
    int A[SIZE];
    int sum1;
    int sum2;
} shm_t;
```

```
int main() {
    shm_t* shm = setup();
    if(fork()==0) {
        int local=0;
        for(int i=0; i<SIZE/2; i++) local += shm->array[i];
        shm->sum1 += local;
        cleanup_and_exit();
    } else {
        int local=0;
        for(int i=SIZE/2; i<SIZE; i++) local += shm->array[i];
        shm->sum2 += local;
        wait(NULL);
    }
    int total = shm->sum1 + shm->sum2;
    cleanup();
    return 0;
}
```

- Both child and the parent process calculate the partial sum independently
- Parent process combine the partial sum from the child to its own calculation to get the total
    - Done only after **wait**

Array Sum Program (Version 2)

```
int main() {
    shm_t* shm = setup();

    int chunks = SIZE/NPROCS;
    for(int i=0; i<NPROCS; i++) {
        if(fork()==0) {
            int local=0;
            int start = i*chunks;
            int end = start+chunk;
            for(int i=start; i<end; i++) local += shm->array[i];

            shm->sum += local;

            cleanup_and_exit();
        }
    }
    for(int i=0; i<NPROCS; i++) wait(NULL);
    cleanup();
    return 0;
}
```

```
typedef struct shm_t {
    int A[SIZE];
    int sum;

} shm_t;
```

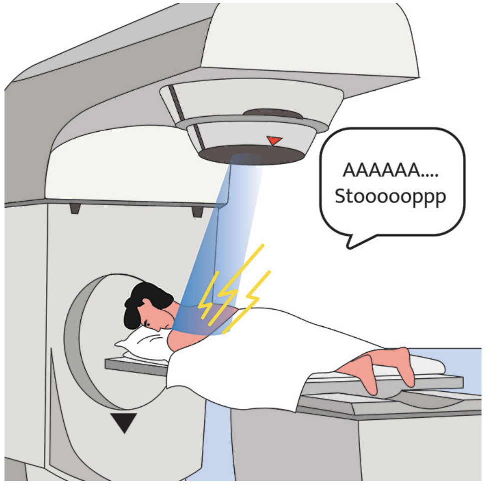
- We want more than one process to participate in the parallel array sum computation
- Children update the global **sum** in the shared memory region
    - Is is correct?
        - **Race condition!**
        - All children are updating the same shared global variable sum

Race Condition

```
shm->sum += local;
```

- Final value of the sum will not be correct as there is a race between reading and writing the counter “sum” by multiple processes
- The line of code shown above is called as critical section
  - Critical section is a line **or** block of code that access shared modifiable data or resource that should be operated on by only one process at a time

Real World Accidents From Race Conditions

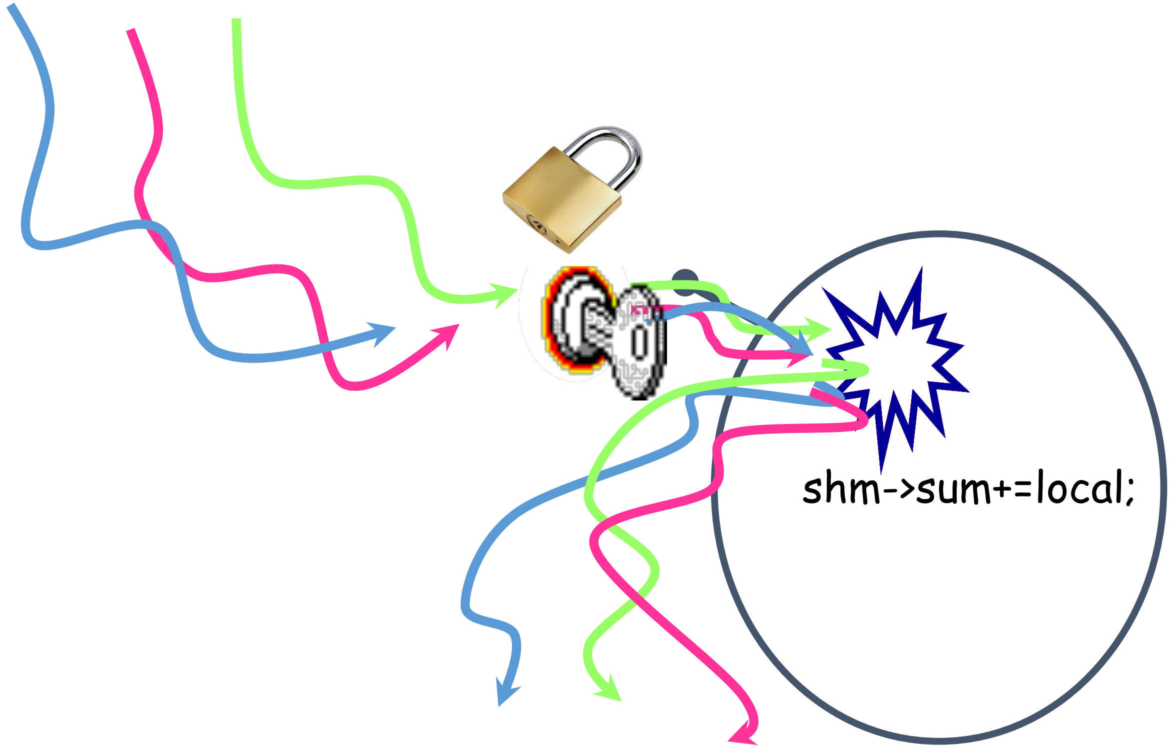


- Therac-25 radiation overdose in 1980s
  - Radiation overdose as the software failed to detect when the operator finished editing due to race condition. It resulted in several deaths and severe injuries
- Northeastern blackout of 2003
  - Race conditions failed to notify the operators about faults occurring in the power grid system
- NASDAQ software glitch in 2012
  - A race condition prevented the delivery of order Facebook IPO confirmations, so those orders were re-submitted repeatedly

Mutual Exclusion

- Mutual exclusion is a property that ensures that there is no race condition by executing the critical section by a single process only at any given time
  - One way to achieve it is by using locks 
  - We will revisit this topic during lectures on concurrency

Visualizing Mutual Exclusion



- Each process must acquire a lock before entering a critical section
- A "Lock" should allow only one process to enter the critical section
- Rest all processes should queue (wait) to get the key
- The process acquiring the lock must release it when exiting the critical section

Naïve Implementation of a Lock (1/3)

```
int value = FREE //inside SHM
```

```
Acquire() {
  Disable interrupts
  if (value == BUSY) {
    sleep
    // Process moved to wait queue

  }
  value = BUSY
  Enable interrupts
}
```

```
Release() {
  Disable interrupts
  if (anyone in wait queue) {
    Move a process into ready queue
  }
  value = FREE
  Enable interrupts
}
```

- Goal: achieving mutual exclusion over a critical section when multiple processes are going to execute that critical section
  - **Let us assume a uniprocessor system for simplicity**
- A naïve way is to let a process complete the execution of a critical section without interrupting it
- Any issues here?
  - The sleeping process has disabled process scheduling!

Naïve Implementation of a Lock (2/3)

```
int value = FREE //inside SHM
```

```
Acquire() {
  Disable interrupts
  if (value == BUSY) {
    Enable interrupts and sleep
    // Process moved to wait queue
    Disable interrupts
  }
  value = BUSY
  Enable interrupts
}
```

```
Release() {
  Disable interrupts
  if (anyone in wait queue) {
    Move a process into ready queue
  }
  value = FREE
  Enable interrupts
}
```

- Solution?
    - The sleeping process must enable interrupts before going to sleep and must disable it after coming out of sleep

Naïve Implementation of a Lock (3/3)

```
int value = FREE //inside SHM
```

```
Acquire() {
  Disable interrupts
  if (value == BUSY) {
    Enable interrupts and sleep
    // Process moved to wait queue
    Disable interrupts
  }
  value = BUSY
  Read a story book?
  Enable interrupts
}
```

```
Release() {
  Disable interrupts
  if (anyone in wait queue) {
    Move a process into ready queue
  }
  value = FREE
  Enable interrupts
}
```

- Any issues?
  - Process acquiring the lock will enjoy an everlasting vacation on the CPU
- Critical section must be as small as possible

Atomic Read-Modify-Write Instructions

- Process updating a variable will have to first “Read” (R) the variable, followed by “Modify” (M) and finally “Store” (S) the updated value so as to be visible to other processes
  - Any issues?
    - P1(Var)→R; P2(Var)→M; P3(Var)→S
      - Overlapping RMS!
- How to fix using hardware support?
  - Hardware can combine RMS as a single instruction for a specific type of variables (**Atomic**)
  - Atomic instructions read a value from memory and write a new value atomically
    - E.g., compare and swap (CAS), atomic increment/decrement, atomic load/store etc.
    - Hardware is responsible for implementing this correctly
  - User code can easily access it both on a uniprocessor and multiprocessors systems

Mutual Exclusion using CAS (1/2)

```
int value = FREE //inside SHM
```

```
Acquire() {
  while (CAS(value, FREE, BUSY) != true);
}
```

```
Release() {
  value = FREE
}
```

- CAS atomically checks if the **value** is **FREE** and set it to **BUSY** if currently **FREE**
  - Returns **true**, otherwise **value** remains unchanged and returns **false**
- **Any issues?**
  - Busy waiting!
    - Process wastes CPU cycles carrying out constant checks until the lock is free
  - What should the process do when it finds the value!=FREE?

Mutual Exclusion using CAS (2/2)

```
int value = FREE //inside SHM
```

```
Acquire() {
  while (CAS(value, FREE, BUSY) != true) {
    sleep();
  }
}
```

```
Release() {
  value = FREE
}
```

- Forcing the process to sleep will move it to the wait queue
- **Any issues?**
  - Sleep for how long?
- **Desirable scenario**
  - If any process fails to acquire the lock then it goes to sleep but is awakened by the process calling the release
    - Can be achieved using Semaphores!

Semaphores

Value of Semaphore	Process-T1	Process-T2
1		
1	call sem_wait()	
0	sem_wait() returns	
0	(crit sect)	
0	call sem_post()	
1	sem_post() returns	

Val	Process-T1	State	Process-T2	State
1		Run		Ready
1	call sem_wait()	Run		Ready
0	sem_wait() returns	Run		Ready
0	(crit sect begin)	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	call sem_wait()	Run
-1		Ready	decr sem	Run
-1		Ready	(sem<0)→sleep	Sleep
-1		Run	*Switch→T0*	Sleep
-1	(crit sect end)	Run		Sleep
-1	call sem_post()	Run		Sleep
0	incr sem	Run		Sleep
0	wake(T1)	Run		Ready
0	sem_post() returns	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	sem_wait() returns	Run
0		Ready	(crit sect)	Run
0		Ready	call sem_post()	Run
1		Ready	sem_post() returns	Run

- A semaphore is an object with an integer **value** (user initialized) that could be manipulate with:
  - **sem_wait**
    - value = value **-1** (atomically!)
    - Blocking (sleep) if value < 0, otherwise non-blocking
  - **sem_post**
    - Non-blocking API
    - value = value **+1** (atomically!)
    - If there are one more processes blocked inside sem_wait then wake **any one** of them
  - Helps in achieving mutual exclusion over a critical section
    - Uses atomic instructions internally

Array Sum Program (Version 2)

```
int main() {
    shm_t* shm = setup();
    sem_init(&shm->mutex, 1, 1);
    int chunks = SIZE/NPROCS;
    for(int i=0; i<NPROCS; i++) {
        if(fork()==0) {
            int local=0;
            int start = i*chunks;
            int end = start+chunk;
            for(int i=start; i<end; i++) local += shm->array[i];
            sem_wait(&shm->mutex);
            shm->sum += local;
            sem_post(&shm->mutex);
            cleanup_and_exit();
        }
    }
    for(int i=0; i<NPROCS; i++) wait(NULL);
    cleanup();
    return 0;
}
```

```
typedef struct shm_t {
    int A[SIZE];
    int sum;
    sem_t mutex;
} shm_t;
```

1) munmap (shm, ...)
2) close (...)
3) shm_unlink (...)
4) **sem_destroy**

- We used a binary semaphore to synchronize the accesses on the sum variable
- Semaphore helped in achieving mutual exclusion over the critical section
  - No more race condition!

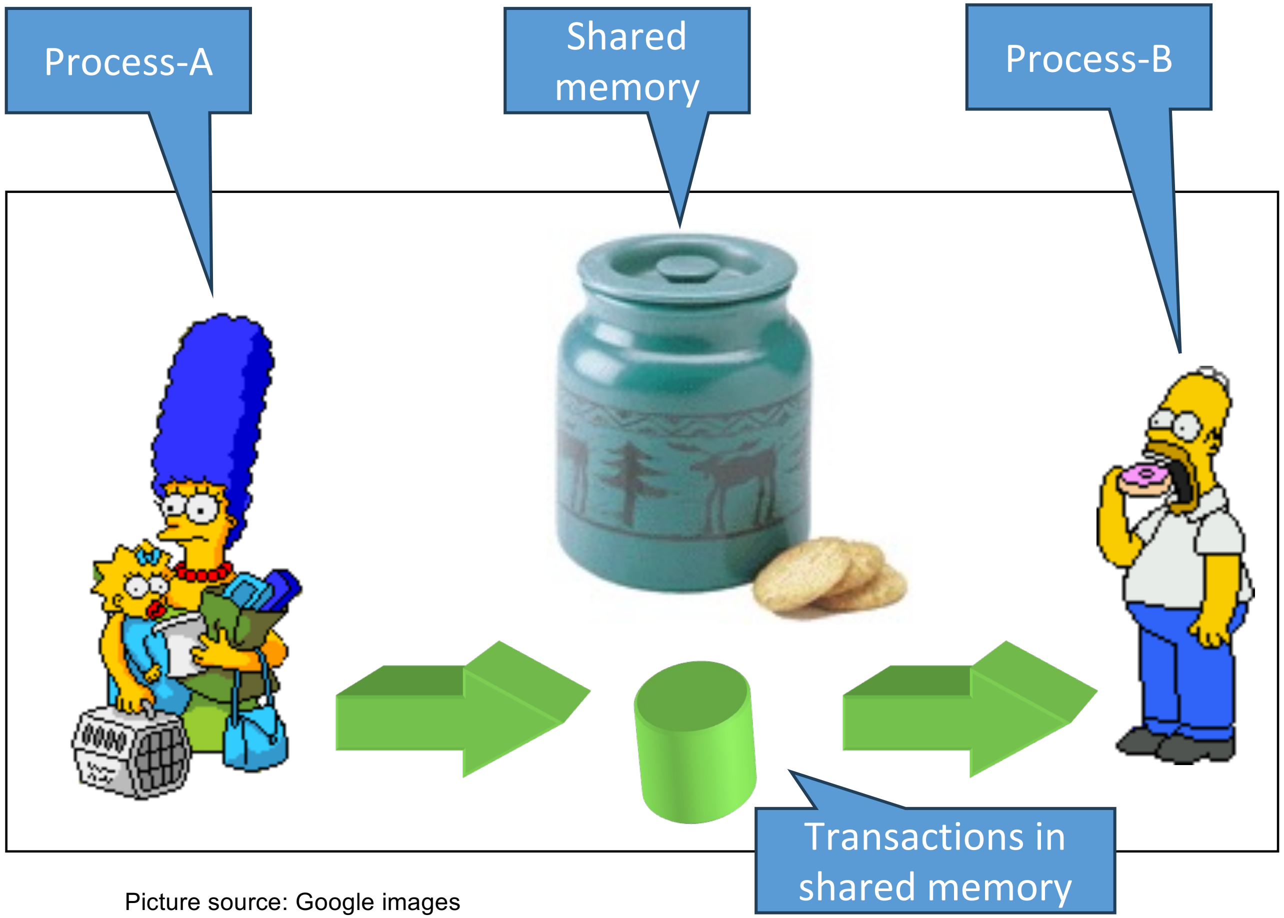
Measuring the Parallel Performance?

- Speedup is used to denote the performance improvement by using multiple processes
  - Speedup = $T_{serial}/T_{parallel}$

Today’s Class

- Shared memory
  - Introduction to parallel computing
  - Semaphores for process synchronization
- Producer consumer problem

The Producer Consumer Problem



Picture source: Google images

- The Simpsons story
  - Homer is fond of eating cookies every time
  - Marge is super busy with her baby and so cannot bake a lot of cookies once
  - The cookie jar is the place to store cookies shared between Marge and Homer
  - Homer picks a cookies from cookie jar if available and then waits for Marge to prepare one for him
  - Marge waits for Homer to eat the cookie from cookie jar and prepares one if no cookie is available
- We need to synchronize between transactions, for example, the producer-consumer scenario described above (or bounded buffer problem)

The Producer Consumer Problem

```
vivek@possum:~/os23$ ./a.out
Marge bake Cookie-1
Homer ate Cookie-1
Marge bake Cookie-2
Homer ate Cookie-2
Marge bake Cookie-3
Homer ate Cookie-3
Marge bake Cookie-4
Homer ate Cookie-4
Marge bake Cookie-5
Homer ate Cookie-5
```

- The Simpsons story
  - Homer is fond of eating cookies every time
  - Marge is super busy with her baby and so cannot bake a lot of cookies once
  - The cookie jar is the place to store cookies shared between Marge and Homer
  - Homer picks a cookies from cookie jar if available and then waits for Marge to prepare one for him
  - Marge waits for Homer to eat the cookie from cookie jar and prepares one if no cookie is available
- We need to synchronize between transactions, for example, the producer-consumer scenario described above (or bounded buffer problem)

The Producer Consumer Problem

```
typedef struct cookiejar_t {
    int cookie;
    int empty;
} cookiejar_t;

cookiejar_t* cookiejar;

int main() {
    cookiejar = setup();
    cookiejar->empty=1;
    if(fork() == 0) { homer(); }
    if(fork() == 0) { marge(); }
    wait(NULL); // wait for Homer process
    wait(NULL); // wait for Marge process
    cleanup();
    return 0;
}
```

1) shm_open (...)
2) ftruncate (...)
3) mmap (...)

1) munmap (...)
2) close (...)
3) shm_unlink (...)

- We will create a shared memory segment of the type cookiejar_t that contains the "cookie" number and a flag "empty" to indicate the availability of a cookie in the cookie jar
- Setup and cleanup of shared memory region works exactly as described in earlier slides

The Producer Consumer Problem

```
void homer() {
    for(int i=0; i<5; i++) {
        while(cookiejar->empty) {
            /*Loop endlessly*/
        }
        printf("Homer ate Cookie-%d\n", cookiejar->cookie);
        cookiejar->empty = 1;
    }
    cleanup_and_exit();
}
```

```
void marge() {
    for(int i=0; i<5; i++) {
        while(!cookiejar->empty) {
            /*Loop endlessly*/
        }
        printf("Marge bake Cookie-%d\n", ++cookiejar->cookie);
        cookiejar->empty = 0;
    }
    cleanup_and_exit();
}
```

```
vivek@possum:~/os23$ ./a.out
Marge bake Cookie-1
^C
vivek@possum:~/os23$ ./a.out
Marge bake Cookie-1
^C
vivek@possum:~/os23$ ./a.out
Marge bake Cookie-1
^C
```

- I tried running the program several times but Homer was simply not eating any cookie despite the fact that Marge had in fact baked one for him..
  - What went wrong?

Compiler Ruined the Show!

```
void homer() {
    for(int i=0; i<5; i++) {
        while(true) {
            /*Loop endlessly*/
        }
        printf("Homer ate Cookie-%d\n", cookiejar->cookie);
        cookiejar->empty = 1;
    }
    cleanup_and_exit();
}
```

```
void marge() {
    for(int i=0; i<5; i++) {
        while(false) {
            /*Loop endlessly*/
        }
        printf("Marge bake Cookie-%d\n", ++cookiejar->cookie);
        cookiejar->empty = 0;
    }
    cleanup_and_exit();
}
```

- Compiler did not know that the value of variable "**empty**" is being changed by another process
- It noticed during compilation that main() first sets empty=1 and then calls the methods homer() and marge() where the variable **empty** was simply being tested inside while condition
- It acts smart and carries out optimization by simply replacing the variable "**empty**" with 1 at all places
- Fix?
    - Declare the variables "empty" as **volatile**

The Producer Consumer Problem

```
void homer() {
    for(int i=0; i<5; i++) {
        while(cookiejar->empty) {
            /*Loop endlessly*/
        }
        printf("Homer ate Cookie-%d\n", cookiejar->cookie);
        cookiejar->empty = 1;
    }
    cleanup_and_exit();
}
```

```
void marge() {
    for(int i=0; i<5; i++) {
        while(!cookiejar->empty) {
            /*Loop endlessly*/
        }
        printf("Marge bake Cookie-%d\n", ++cookiejar->cookie);
        cookiejar->empty = 0;
    }
    cleanup_and_exit();
}
```

```
vivek@possum:~/os23$ ./a.out
Marge bake Cookie-1
Homer ate Cookie-1
Marge bake Cookie-2
Homer ate Cookie-2
Marge bake Cookie-3
Homer ate Cookie-3
Marge bake Cookie-4
Homer ate Cookie-4
Marge bake Cookie-5
Homer ate Cookie-5
```

- **volatile** seems to have fixed the problem?
  - It is an incorrect solution
  - What happens when more than one processes are racing for updating the variable "empty"?
    - **Race condition!**

Semaphores to the Rescue!

```
typedef struct cookiejar_t {
    int cookie;
    sem_t jar_empty;
    sem_t jar_full;
} cookiejar_t;

cookiejar_t* cookiejar;

int main() {
    cookiejar = setup();
    cookiejar->empty=1;
    sem_init(&cookiejar->jar_empty, 1, 1);
    sem_init(&cookiejar->jar_full, 1, 0);
    if(fork() == 0) { homer(); }
    if(fork() == 0) { marge(); }
    wait(NULL); // wait for Homer process
    wait(NULL); // wait for Marge process
    sem_destroy(&cookiejar->jar_empty);
    sem_destroy(&cookiejar->jar_full);
    cleanup();
    return 0;
}
```

- We will declare two semaphore variables inside the shared memory region so as they can be shared by both Marge and Homer processes
- Main process will call the **init** and **destroy** APIs on these semaphore objects

Semaphores to the Rescue!

```
void homer() {
    for(int i=0; i<5; i++) {
        sem_wait(&cookiejar->jar_full);
        printf("Homer ate Cookie-%d\n", cookiejar->cookie);
        sem_post(&cookiejar->jar_empty);
    }
    cleanup_and_exit();
}
```

```
void marge() {
    for(int i=0; i<5; i++) {
        sem_wait(&cookiejar->jar_empty);
        printf("Marge bake Cookie-%d\n", ++cookiejar->cookie);
        sem_post(&cookiejar->jar_full);
    }
    cleanup_and_exit();
}
```

```
vivek@possum:~/os23$ ./a.out
Marge bake Cookie-1
Homer ate Cookie-1
Marge bake Cookie-2
Homer ate Cookie-2
Marge bake Cookie-3
Homer ate Cookie-3
Marge bake Cookie-4
Homer ate Cookie-4
Marge bake Cookie-5
Homer ate Cookie-5
```

- At the start, the value of semaphores **jar_full** and **jar_empty** was set to **0** and **1**, respectively
- Homer process upon creation will block in sem_wait as the value of jar_full was initially 0 (decremented to -1)
- Marge upon activation will decrement jar_empty to 0 and will not block. It will bake a cookie, increment the jar_full (now its zero), and wake up Homer from sem_wait. Finally, it will block inside sem_wait after decrementing jar_empty (now its -1)
- Homer will awake from sem_wait, eat the cookie, increment the jar_empty semaphore (now its 0), and wake up Marge from sem_wait. Finally, it will block inside sem_wait after decrementing jar_full (now its -1)
- And the cookie business continues for five times..
- We will revisit the topic of mutual exclusion during lectures on concurrency (multithreading), where we will again discuss it in depth!

Next Lecture

- IPC in distributed memory
  - Last remaining topic in IPC
- Quiz-2 on Thursday
  - Syllabus: Lectures 05-09
- Assignment-2 will be released on Friday (13th Sep)