A System Using MMU Table

```
void scheduler() {
    while(true) {
        lock(process_table);
        foreach(Process p: scheduling_algorithm(process_table)) {
            if(p->state != READY) {
                continue;
            }
            p->state = RUNNING;
            unlock(process_table);
            swtch_mmu_table(scheduler_process, p);
            swtch(scheduler_process, p);
            // p is done for now..
            lock(process_table);
        }
        unlock(process_table);
    }
}
```

```
void scheduler() {
    while(true) {
        lock(process_table);
        foreach(Process p: scheduling_algorithm(process_table))
{
            if(p->state != READY) {
                continue;
            }
            p->state = RUNNING;
            unlock(process_table);
            swtch_mmu_bbRegisters(scheduler_process, p);
            swtch(scheduler_process, p);
            // p is done for now..
            lock(process_table);
        }
        unlock(process_table);
    }
```

- swtch_mmu_bbRegisters (p1, p2)
  - Simply save the content of base/bound registers of currently executing P1 process into its PCB, and load the base/bound registers from P2's PCB

The Act of Good Parenting

```
int create_process_and_run(char* command) {
    int status = fork();
    if(status < 0) {
        printf("Something bad happened\n");
        exit(0);
    } else if(status == 0) {
        printf("I am the child (%d)\n",getpid());
    } else {
        int ret;
        int pid = wait(&ret);
        if(WIFEXITED(ret)) {
          printf("%d Exit =%d\n",pid,WEXITSTATUS(ret));
        } else {
          printf("Abnormal termination of %d\n",pid);
        }
        printf("I am the parent Shell\n");
    }
    return 0;
}
```

- **wait** and **waitpid** allows the parent process to block until the child process terminates
  - **wait** will block only for the first child, whereas **waitpid** can be used for a specific child
  - Returns the child's PID
  - Used for retrieving exit status from child
- Will there be deterministic execution of printfs from parent and child processes (notice there is no sleep)?
- Good parents **wait** for their children to avoid making them **zombies** (terminated child's exit code remaining in the process table)
- However, **orphaned** children outliving their parent's lifetime are adopted by the mother-of-all-processes (**init**)

```
int create_process_and_run(char* command) {
    int status = fork();
    if(status < 0) {
        printf("Something bad happened\n");
        exit(0);
    } else if(status == 0) {
        int status2 = fork();
        if(status2<0) printf("Kindness failed\n");
        else if (status2 == 0) {
           printf("Child will not live like Zombie\n");
        } else {
           _exit(0);
        }
    } else {
        printf("I am the parent Shell\n");
    }
    return 0;
}
```

```
void scheduler() {
    while(true) {
        lock(process_table);
        foreach(Process p: scheduling_algorithm(process_table)) {
            if(p->state != READY) {
                continue;
            }
            p->state = RUNNING;
            unlock(process_table);
            swtch(scheduler_process, p);
            // p is done for now..
            lock(process_table);
        }
        unlock(process_table);
    }
}
```

- Process scheduling algorithm plays an important role in the design of operating system
- Different algorithms are chosen according to the need

Programming Signals

```
static void my_handler(int signum) {
  static int counter = 0;
  if(signum == SIGINT) {
    char buff1[23] = "\nCaught SIGINT signal\n";
    write(STDOUT_FILENO, buff1, 23);
    if(counter++==1) {
      char buff2[20] = "Cannot handle more\n";
      write(STDOUT_FILENO, buff2, 20);
      exit(0);
    }
  } else if (signum == SIGCHLD) {
    char buff1[23] = "Caught SIGCHLD signal\n";
    write(STDOUT_FILENO, buff1, 23);
  }
}
```

```
int main() {
  struct sigaction sig;
  memset(&sig, 0, sizeof(sig));
  sig.sa_handler = my_handler;
  sigaction(SIGINT, &sig, NULL);
  sigaction(SIGCHLD, &sig, NULL);
  int n;
  while(1) {
    printf("Input i: \n");
    scanf("%d",&n);
    if(fork()==0) {
      printf("Fib(%d) = %d\n",n,fib(n));
      exit(0);
    }
    else wait(NULL);
  }
  return 0;
}
```

Why Fib(20) is calculated twice?

```
vivek@possum:~/os23$ ./a.out
Input i:
10
Fib(10) = 55
Caught SIGCHLD signal
Input i:
20
Fib(20) = 6765
Caught SIGCHLD signal
Input i:
^C
Caught SIGINT signal
Fib(20) = 6765
Caught SIGCHLD signal
Input i:
^C
Caught SIGINT signal
Cannot handle more
```

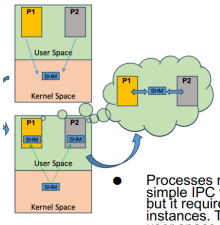
- sigaction is used to change the default action associated with a signal
  - Except SIGKILL and SIGSTOP
- User can assign his own signal handler method that would get invoked when the assigned signal is received by the process
- Only asynchronous safe functions should be called inside signal handler (e.g., never call printf)
  - https://man7.org/linux/man-pages/man7/signal-safety.7.html

Programming With pipe and dup

```
int main() {
    int fd[2], status;
    pipe(fd);
    if(fork() == 0) {
        /* Child process */
        close(fd[0]);
        dup2(fd[1], STDOUT_FILENO);
        char buff[] = "Hello my dear good Parent";
        printf("%s", buff);
        exit(0);
    }
    /* Parent process */
    close(fd[1]);
    dup2(fd[0], STDIN_FILENO);
    char buff[100];
    read(fd[0], buff, sizeof(buff));
    printf("My obedient child says: %s\n", buff);
    wait(NULL);
    return 0;
}
```

- dup2 can be used to duplicate a file descriptor
  - E.g., duplicate one of the end of the pipe as STDOUT or STDIN
    - Duplicating to STDOUT will cause printf to print to the pipe instead of the STDOUT
- Used by the Shell when we pipe the output of one command to another command

Last Lecture



- Processes running inside a single machine can use simple IPC techniques such as signaling or pipes, but it requires help from OS for every signaling instances. There is always a transition between user space and kernel space (overheads)
- Shared memory is the fastest form of IPC in which processes can asynchronously write to a shared region of memory without the need for switching to kernel space
- A semaphore is an object with an integer **value** that is used for achieving mutual exclusion over a critical section

Array Sum Program

```
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!

The Hello World Program in MPI (3/3)

```
// the header file containing MPI APIs
#include <mpi.h>
int main(int argc, char **argv) {
    // Initialize the MPI runtime
    MPI_Init(&argc, &argv);
    int rank, nprocs;
    // Get the total number of processes in MPI_COMM_WORLD
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    // Get the rank of this process in MPI_COMM_WORLD
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("My rank is %d in world of size %d\n", rank, nprocs);
    // Terminate the MPI runtime
    MPI_Finalize();
    return 0;
}
```

- Each of the processes will get a unique rank (id) in the group MPI_COMM_WORLD that can be enquired using the MPI_Comm_rank API
- MPI rank can be used to decide what will execute on which process

There is no user data being exchanged in this program. We will see message exchange in the next slides. Each process will simply print their rank and total number of processes invoked by the user

Parallel Array Sum Using MPI (4/4)

```
int main(int argc, char **argv) {
    int rank=0, nproc=4;
    MPI_Init(&argc, &argv);
    // 1. Get to know your world
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    int array[SIZE]; // initialized and assume (SIZE % nproc = 0)
    // 2. calculate local sum
    int my_sum = 0, chunk = SIZE/nproc;
    for (int i=rank*chunk; i<(chunk+1)*rank; i++) my_sum += array[i];
    // 3. All non-root processes send result to root processes (rank=0)
    if(rank > 0) {
        MPI_Send(&my_sum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    else { // executed only at rank=0
        int total_sum = my_sum, tmp;
        for(int src=1; src<nproc; src++) {
          MPI_Recv(&tmp, 1, MPI_INT, src, 0, MPI_COMM_WORLD, NULL);
          total_sum += tmp;
        }
    }
    MPI_Finalize();
}
```

- Root rank will use (nproc-1) number of MPI_Recv API to receive the message being sent by (nproc-1) number of processes
- Parameters to MPI_Recv are exactly same as in MPI_Recv with few minor changes
  - The first parameter is the buffer used to store the message receive from the destination. It must be of the same size and datatype as in the sender side
  - The fourth parameter is the rank of the sender
  - You can ignore the last parameter and simply pass it as NULL

A System Using Paging

```
void scheduler() {
    while(true) {
        lock(process_table);
        foreach(Process p: scheduling_algorithm(process_table)) {
            if(p->state != READY) {
                continue;
            }
            p->state = RUNNING;
            unlock(process_table);
            swtch_pagetable_base_registers(scheduler_process, p);
            swtch(scheduler_process, p);
            // p is done for now..
            lock(process_table);
        }
        unlock(process_table);
    }
}
```

```
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 j=start; j<end; j++) local += shm->array[j];
            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[NPROCS];
    sem_t mutex;
} shm_t;
```

Array Sum using Pthread

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE]; // Initialization code elided
int array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

```c
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i] , NULL);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Money Transaction Between Accounts

```c
class Account {
  int id;
  double balance;
  pthread_mutex_t m =
          PTHREAD_MUTEX_INITIALIZER;
  void debit(double amount);
  void credit(double amount);
};
```

```c
class Bank {
  void fund_transfer() {
    Accounts numAccounts[N];
    Transfer pending[TOTAL];
    parallel_for(int i=0; i<TOTAL; i++) {
      pending[i].run();
    }
  }
};
```

```c
class Transfer {
  Account source, destination;
  double amount;
  void run() {
    Account a1, a2;
    if(source.id < destination.id) {
      a1 = source; a2 = destination;
    } else {
      a1 = destination; a2 = source;
    }
    a1.lock(); a2.lock();
    source.debit(amount);
    destination.credit(amount);
    a2.unlock(); a1.unlock();
  }
};
```

Deadlock resolved using lock ordering

Array Sum using Pthread (Version-2)

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE], result=0;
pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER;
void array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  pthread_mutex_lock(&m);
  result += sum;
  pthread_mutex_unlock(&m);
}

typedef struct {
  int low;
  int high;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  array_sum(t->low, t->high);
  return NULL;
}
```

Race condition is fixed using mutual exclusion

```c
int main(int argc, char *argv[]) {
  if (SIZE < 1024) {
    array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i]);
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Producer Consumer using Pthreads

```
1. pthread_mutex_lock(&mutex);
2. while(task_queue_size() == 0)
3.   pthread_cond_wait(&cond, &mutex);
4. }
5. task = pop_task_queue();
6. pthread_mutex_unlock(&mutex);
7. execute_task (task);
```

Consumer(s)

```
1. pthread_mutex_lock(&mutex);
2. int queue_size = task_queue_size();
3. push_task_queue(&task);
4. if(queue_size == 0) {
5.   pthread_cond_broadcast(&cond);
6. }
7. pthread_mutex_unlock(&mutex);
```

Producer

- pthread_cond_**wait** causes the current thread to relinquish the CPU and wait until another thread invokes the **signal** or the **broadcast**
- Upon call for **wait**, the thread releases ownership of the mutex and waits until another thread signals the waiting threads

Creating File Without Journaling

- Find free data block(s)
- Find free inode entry
- Find directory entry insertion point

---

1. Write map (i.e., mark used)
2. Write inode entry to point to block(s)
3. Write directory entry to point to inode