Laboratory work in TDDI04
Pintos Assignment 4

Viacheslav Izosimov
2009-03-23
viaiz@ida.liu.se

General Description

Lab 4: “Execution, termination and synchronization of user programs”
- Execution of several user programs
- Termination of a user program
- Synchronization of shared data structures
- *Wait* system call

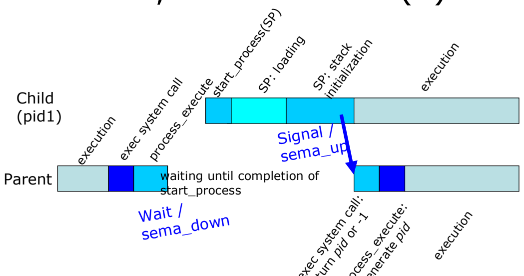
We will go through many issues one more time!

Main Goals

Provide synchronization for multiple programs
Provide synchronization between programs

File synchronization: Not yet addressed. It is a part of Lab 5

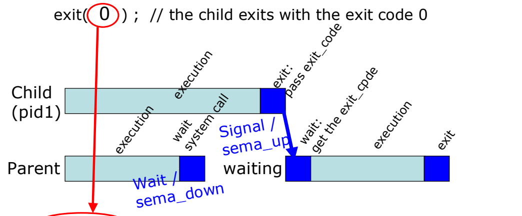
Exec, Exit and Wait (1)



pid1 = exec(“nasty_program”);

Exec, Exit and Wait (2)

exit(0) ; // the child exits with the exit code 0



exit_code = wait(pid1) ; // the parent waits for a child process

Note that a parent can have several children!

Exec, Exit and Wait (3)

pid_t **exec** (const char **cmd_line*)

Runs the executable whose name is given in *cmd_line*, passing any given arguments, and returns the new process’s program *id* (*pid*)

Must return *pid* -1, if the program cannot **load** or **run** for any reason (!)

Exec, Exit and Wait (4)

void **exit** (int *status*)

Terminates the current user program, returning the exit code *status* to the kernel.

status of 0 indicates success and nonzero values indicate errors

Remember to free all the resources that will be not needed anymore.

Exec, Exit and Wait (5)

int **wait** (pid t *pid*)

Provides synchronization between user programs. "Parent" process waits until its *pid*-"child" process dies (if the child is still running) and receives the "child" exit code.
If the child has been finished, *wait()* should return child's exit value without waiting.

Seems to be difficult…

Exec

pid = process ID
tid = thread ID

The Pintos kernel itself has to be associated with *pid*!

Add your implementation of *exec()* functionalities into *process_execute*() and *process_start()* in process.c

Your *Exec*() system call in syscall.c {	*process_execute*() {	*start_process*() {
…	**…**	loading – **DONE!**
f->eax =	*tid* = thread_create	initialization – **DONE!**
process_execute	**…**	putting program arguments into stack
}	generate *pid* from *tid;*	signal to
	wait until *start_process*();	*process_execute*
	return *pid* or -1	}
	}	

pid = -1, if the program cannot **load** or **run** for any reason.
Use an array or a list to keep track of *pid*:s.
pid might equal *tid*, because we have only one thread per process.
Limit the number of user programs (t.ex. 64 or 128).

The Pintos kernel itself has to be associated with *pid*!

threads/init.c

Initialization

Then call *process_init*() function somewhere here

```
/* Pintos main program. */
int
main (void)
{
  char **argv;

  /* Clear BSS and get machine's
  ram_init ();

  /* Break command line into arg
  argv = read_command_line ();
  argv = parse_options (argv);

  /* Initialize ourselves as a t
     then enable console locking
  thread_init ();
  console_init ();

  /* Greet user. */
  printf ("Pintos booting with %

  /* Initialize memory system. *
  palloc_init ();
  malloc_init ();
  paging_init ();

  /* Segmentation. */
#ifdef USERPROG
  tss_init ();
  gdt_init ();
#endif

  /* Initialize interrupt handle
  intr_init ();
  timer_init ();
  kbd_init ();
  input_init ();
#ifdef USERPROG
  exception_init ();
  syscall_init ();
#endif

  /* Start thread scheduler and
  thread_start ();
  serial_init_queue ();
```

Exit (1)

The most suitable place for *Exit()* functionalities is in your implementation of systems calls in syscall.c

Your *Exit*() system call in syscall.c {	*thread_exit*() {	*process_exit*() {
get exit code from user;	**…**	**…**
save the exit code if needed	*process_exit*	clean up program's resources;
thread_exit	}	}
}		

Exit() must return -1 to the "parent" program if something is wrong, for example, if the child has caused a memory violation.
You should take care of it!

Clean up program's resources before the exit!

printf("%s: exit(%d)\n", *thread-name*, *thread-exit-value***)** before any exit.
(This is needed for testing purposes.)

Exit (2)

```
/* Free the current process's resources. */
void
process_exit (void)
{
  struct thread *cur = thread_current ();
  uint32_t *pd;

  /* Destroy the current process's page directory and switch back
     to the kernel-only page directory. */
  pd = cur->pagedir;
  if (pd != NULL)
    {
      /* Correct ordering here is crucial.  We must set
         cur->pagedir to NULL before switching page directories,
         so that a timer interrupt can't switch back to the
         process page directory.  We must activate the base page
         directory before destroying the process's page
         directory, or our active page directory will be one
         that's been freed (and cleared). */
      cur->pagedir = NULL;
      pagedir_activate (NULL);
      pagedir_destroy (pd);
    }
}
```

Wait

Once you get *pid*, just call *process_wait*()
(located in process.c) from *Wait()* system call:

```
/* Waits for thread TID to die and returns its exit status.  If
   it was terminated by the kernel (i.e. killed due to an
   exception), returns -1.  If TID is invalid or if it was not a
   child of the calling process, or if process_wait() has already
   been successfully called for the given TID, returns -1
   immediately, without waiting.

   This function will be implemented in problem 2-2.  For now, it
   does nothing. */
int
process_wait (tid_t child_tid UNUSED)
{
  return -1;
}
```

Steps to accomplish *wait()*:

1. **Wait** until the *exit code* of child *pid* is available
2. **Get** the *exit code* and **remove it from the system**
3. **Return** the *exit code* (or -1 if something is wrong)

Situations with Wait (1)

"Parent" exits without calling *wait()* while the "child" is still running

"Child" exits before the "parent" and:

"parent" calls *wait()* afterwards, or

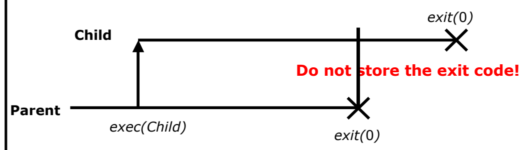
"parent" will exit without calling *wait()*.

"Parent" calls *wait()* before the "child" exits.

All the situations above under the condition that the child does not exit normally.

Situations with Wait (2)

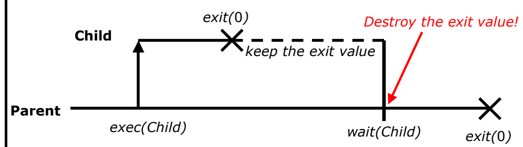
"Parent" exits without calling *wait()* while the "child" is still running



Situations with Wait (3)

"Child" exits before the "parent" and:

"parent" calls *wait()* afterwards

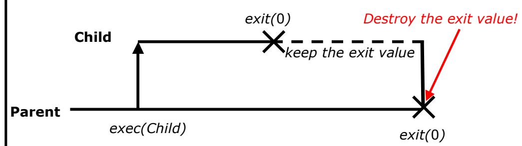


wait() returns child's exit value without waiting

Situations with Wait (4)

"Child" exits before the "parent" and:

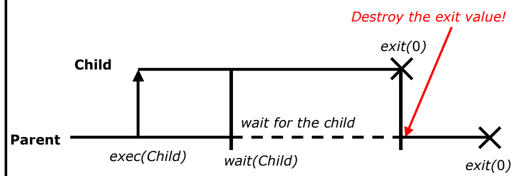
"parent" will exit without calling *wait()*.



You should keep child's exit value until the parent exits
(since the child doesn't know if the parent calls *wait*() later on)

Situations with Wait (5)

"Parent" calls *wait()* before the "child" exits.



the parent waits for its child…

Situations with Wait (6)

All the situations above under the condition that the child does not exit normally.

To Be Remembered…

Parts of the functions accessing shared resources **must** be thread safe, e.g. employ synchronization techniques such as locks and semaphores.

Particularly, access to global objects and data **must** be synchronized.

Only **one** thread can have access to the **console** at a
Other threads must wai
completion of reading/wr

Exercise 1

```
bool allocate(struct content *list[], int size)
{
  …
  for (int i = 0; i < size; ++i)
  {
     if (list[i] == NULL)
        break;
  }
  …
  lock_list();
  list[i] = malloc(sizeof(struct content));
  unlock_list();
  …
  return true;
}
```

Exercise 2

```
struct lock life_lock; /* global lock */

void incr_day_cnt(struct stlife * life)
{
   lock_aquire(life_lock);
   life->day_cnt++;
   lock_release(life_lock);
}

void incr_bicycle_cnt(struct stlife * life)
{
   lock_aquire(life_lock);
   life->bicycle_cnt++;
   lock_release(life_lock);
}
```

Exercise 3

```
struct lock life_lock; /* global lock */

void init_life(struct lifes * life_list) {
   struct stlife *life;
   int life_ind = get_life_ind(life_list);
   lock_aquire(life_lock);
   life = malloc (sizeof *life);
   life->day_cnt = 0;
   life->bicycle_cnt = 0;
   life->CSN_cnt = 0;
   …
   init_lock(life->bicycle_lock);
   …
   life_list[life_ind] = life;
   lock_release(life_lock);
}
```

```
void new_life(struct stlife * life) {
   lock_aquire(life_lock);
   life->day_cnt = 0;
   life->bicycle_cnt = 0;
   life->CSN_cnt = 0;
   …
   lock_release(life_lock);
}

void incr_day_cnt(struct stlife * life) {
   lock_aquire(life->day_lock);
   life->day_cnt++;
   lock_release(life->day_lock);
}
```

Exercise 4

```
int process_wait (pid_t child_pid) {
    lock_acquire(pidListLock);
    pid_t parentId = get_pid_id(thread_current()->tid);
    struct processListItem * parent = &processList[parentId];
    struct processListItem * child = &processList[child_pid];
    if(child->parent == parentId) {
        if(!(child->exited)){
            parent->isSleeping = 1;
            parent->waitingForChild = child_pid;
            cond_wait(pidCond, pidListLock);
        }
        reset_process(child_pid);
        lock_release(pidListLock);
        return child->exit_value;
    } else return -1;
}
```

Test (1)

pintos -v -k --qemu -p ../../examples/parent -a parent -p ../../examples/child -a child -- -f -q run parent

```c
/* parent.c */
#include <syscall.h>
#include <stdlib.h>
#include <stdio.h>
#define CHILDREN 4
#define DEPTH 3
int main(int argc, char* argv[]){
  int i;
  int pid[CHILDREN];
  int depth = DEPTH - 1;
  char cmd[10];
  if (argc == 2)
    depth = atoi(argv[1]) - 1;
    for(i = 0; i < CHILDREN; i++)  {
    if (depth)
      snprintf(cmd, 10, "parent %i", depth);
    else
      snprintf(cmd, 10, "child %i", i);
    printf("%s\n", cmd);
    pid[i] = exec(cmd);
  }
  for(i = 0; i < CHILDREN; i++)    {
      wait(pid[i]);
  }
  exit(0);
}
```

```c
/* child.c */
#include <syscall.h>
#include <stdio.h>
int main (int argc, char* argv[]){
  int i;
  if (argc != 2)
    return 0;
  for(i = 0; i < 20000; i++)  {
     int a = (i * i) + (i * i);
     int b = i;
     i = a; a = b; i = b;
 }
 printf("PASS Lab %s ON Time.\n", argv[1]);
 return 0;
}
```

Test (2)

pintos -v -k --qemu -p ../../examples/longrun -a longrun -p ../../examples/dummy -a dummy -- -f -q run 'longrun 10 1000'

```c
/* Start a lot of processes and let them finish
* to test if we eventually run out of process slotes. */
#include <syscall.h>
#include <stdlib.h>
#include <stdio.h>
#define SIMUL 10  /* simultaneously running */
#define TOTAL 200 /* totally started */
int main(int argc, char* argv[]){
  char cmd[15];
  int pid[50];
  int i, j;
  int total;
  int simul;
  if (argc == 3) total = atoi(argv[2]);
  else total = TOTAL;
  if (argc == 2 || argc == 3)
    simul = atoi(argv[1]);
  else simul = SIMUL;
  for (j = 0; j < total / simul; ++j)  {
    for (i = 0; i < simul; ++i)    {
       snprintf(cmd, 15, "dummy %i",
         j * simul + i);
       pid[i] = exec(cmd);
    }
    for (i = 0; i < 50; ++i) wait(pid[i]);
 }
 return 0;
}
```

```c
/* A small dummy process
* that just uses up a process slot
* in the long runtime test */

#include <stdlib.h>
int main(int argc, char* argv[]){
  if (argc != 2)
    return 0;
   return atoi(argv[1]);
}
```

Test (3)

The following checks should pass when you run gmake check:

Different exec-tests:
tests/userprog/exec-once
tests/userprog/exec-arg
tests/userprog/exec-multiple
tests/userprog/exec-missing
tests/userprog/exec-bad-ptr

Wait-tests:
tests/userprog/wait-simple
tests/userprog/wait-twice
tests/userprog/wait-killed
tests/userprog/wait-bad-pid

Conclusion (1)

Lab 4, probably, is the most important lab during this course

- Execution of several user programs
- Termination of a user program
- Synchronization of shared data structures
- *Wait* system call

Always think about concurrency and correctness!

Complete it before 27th of April!!!

Conclusion (2)

In this course you do the first "real" programming

Learning of handing complex programming tasks

Self-management training

Training of planning skills

Working with a pile of extensive documentation

And, last but not least, understanding of the basic concepts of operating systems

Conclusion (3)

Do not wait until the summer vacation!
Complete your assignments now!