Pintos 프로젝트

# 프로젝트 소개

핀토스 프로젝트는 스탠포드 대학에서 만들어진 교육용 운영체제로 Thread, Process, System Call, Priority Scheduling, 그리고 Virtual Memory와 같은 운영체제 기능을 구현하는 프로젝트입니다.

# 코드 (Process 관련)

|  |  |
| --- | --- |
| 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. \*/  struct thread\* C;  file\_close(cur->Excuting\_file);  for(;cur->opening\_num>1;cur->opening\_num--)//close all file  {  close(cur->opening\_num);  }  pd = cur->pagedir;  if (pd != NULL)  {  cur->pagedir = NULL;  pagedir\_activate (NULL);  pagedir\_destroy (pd);  }  } | int process\_wait (tid\_t child\_tid UNUSED) //child\_tid check  {  struct thread\* cur=thread\_current();  struct thread\* Target\_child=ret\_thread(child\_tid);  if(Target\_child==NULL||Target\_child->parent\_ptr!=thread\_current())  {return -1;}//child\_tid is not child of the process  if(Target\_child->wp\_sema==1)  { int ret=Target\_child->exit\_state;  sema\_up(&Target\_child->wait\_parent);  sema\_init(&cur->wait\_child,0);  Target\_child->is\_parent\_waiting=1;  sema\_down(&cur->wait\_child);  return ret;}  else  {  sema\_init(&cur->wait\_child,0);  Target\_child->is\_parent\_waiting=1;  sema\_down(&cur->wait\_child);  return cur->exit\_state;  }  } |

# 설명

# 정상적인 상황에서 모든 프로세스의 실행 순서는 임의로 실행되고 각 프로세스는 실행을 마치면 자원을 반납하고 실행을 종료합니다. 그리고 자식 프로세스를 생성하는 경우 자식 프로세스가 부모 프로세스보다 먼저 종료되고 부모 프로세스는 자식 프로세스가 먼저 종료될 때까지 대기했다가 프로세스 테이블에서 자식 프로세스를 삭제하는 과정을 거칩니다. 하지만 부모 프로세스가 자식 프로세스를 프로세스 테이블에서 지우지 않고 exit 해버린다면 좀비 프로세스가 되어 오류를 발생시킬 수 있습니다. 또한, 부모 프로세스가 자식 프로세스가 먼저 exit 한다면 자식 프로세스는 고아프로세스가 되어 오류를 발생시킬 수 있습니다.이를 방지하고 부모 자식 프로세스가 정상적으로 종료되게 하도록 Semaphore를 이용하여 자식은 부모가 wait 하기 전까지 종료되지 않도록 lock을 걸고 부모 또한 모든 자식이 exit 할 때까지 대기하도록 하였습니다.

# ※ 핀토스는 멀티 스레드를 지원하지 않아 스레드와 프로세스가 같은 기능을 합니다.

# 코드 (Demand Paging 관련)

|  |  |
| --- | --- |
| static bool  load\_segment (struct file \*file, off\_t ofs, uint8\_t \*upage,  uint32\_t read\_bytes, uint32\_t zero\_bytes, bool writable)  {  ASSERT ((read\_bytes + zero\_bytes) % PGSIZE == 0);  ASSERT (pg\_ofs (upage) == 0);  ASSERT (ofs % PGSIZE == 0);  file\_seek (file, ofs);  struct thread\* cur=thread\_current();  while (read\_bytes > 0 || zero\_bytes > 0) //4KB load per loop  {  /\* Calculate how to fill this page.  We will read PAGE\_READ\_BYTES bytes from FILE  and zero the final PAGE\_ZERO\_BYTES bytes. \*/  struct page\_table\_entry\* p=(struct page\_table\_entry\*)malloc(sizeof(struct page\_table\_entry));  size\_t page\_read\_bytes = read\_bytes < PGSIZE ? read\_bytes : PGSIZE;  size\_t page\_zero\_bytes = PGSIZE - page\_read\_bytes;  p->Ref\_file=file;  p->read\_bytes=page\_read\_bytes;  p->zero\_bytes=page\_zero\_bytes;  p->addr=upage;  p->ofs=ofs;  p->writable=writable;  p->mapid=0;  p->swaped=0;  /\* Get a page of memory. \*/  hash\_insert(&cur->supple\_page,&p->h\_elem);  /\* Advance. \*/  read\_bytes -= page\_read\_bytes;  zero\_bytes -= page\_zero\_bytes;  upage += PGSIZE;  ofs+=page\_read\_bytes;  }  return true;  } | static void  page\_fault (struct intr\_frame \*f)  {  bool not\_present; /\* True: not-present page, false: writing r/o page. \*/  bool write; /\* True: access was write, false: access was read. \*/  bool user; /\* True: access by user, false: access by kernel. \*/  void \*fault\_addr; /\* Fault address. \*/  /\* Determine cause. \*/  not\_present = (f->error\_code & PF\_P) == 0;  write = (f->error\_code & PF\_W) != 0;  user = (f->error\_code & PF\_U) != 0;  if((!not\_present&&write)||is\_kernel\_vaddr(fault\_addr)&&user)  { exit(-1);}  else  {  struct page\_table\_entry\* cur=page\_lookup(fault\_addr);  struct thread\* c=thread\_current();  if(cur==NULL)  {  if(fault\_addr>f->esp-PGSIZE)  {  if(!user&&(fault\_addr<c->k\_esp-PGSIZE)||c->stack\_num>=2048)  {  exit(-1);}  expand\_stack();  }  else{  exit(-1);  }  }  else  {  load\_from\_file(cur);  }  }  } |

# 설명

기본 핀토스는 프로그램이 실행될 때 모든 세그먼트에 대한 물리페이지가 할당되었습니다. 이처럼 실행 시 모든 세그먼트에 대해서 물리페이지(= RAM)를 할당할 경우 메모리 자원을 낭비하게 됩니다. 이를 해결하기 위해서 프로그램을 load 할 때 세그먼트 데이터를 메모리에 직접 할당하지 않고 page table에만 저장하였습니다. 이후 프로세스를 실행하면서 해당 프로세스 페이지 중 프레임에 할당되지 않을 페이지를 참조할 시 page\_fault 함수가 실행되고 해당 페이지가 유효하다면 load\_from\_file 함수를 이용하여 Disk로부터 데이터를 읽고 비어있는 메모리를 할당합니다. 비어있는 메모리 주소를 가져올 때는 Race Condition 문제를 해결하기 위해 lock을 활용하였습니다. 빈 메모리가 없는 경우 Second-Chance 알고리즘을 이용하여 Swap을 통해 빈 메모리를 만들어 주었습니다.